XML BASED SYSTEM FOR UPDATING A DOMAIN MODEL AND GENERATING A FORMATTED OUTPUT

Copyright Statement

Technical Field

This invention relates, in general, to object oriented programming systems, and more particularly to a system for updating a domain model and generating a formatted output.

Background Art

Hyper Text Markup Language or HTML is synonymous with the Internet and the World Wide Web (WWW). HTML allows structural markup of WWW or 'Web' documents. XML or eXtensible Markup Language is well known in the art as a document markup language which offers human-readable semantic markup and is also machine-readable. As a result, XML provides the capability to create, parse and process networked data.

XML documents are composed of entities, which are storage units containing text and/or binary data. Text is composed of character streams that form both the document's character data content and the document's meta-data markup. The markup describes the document's storage layout and logical structure. XML also provides a markup mechanism to impose constraints on the storage layout and logical structure of documents, and it provides mechanisms that can be used for strong typing.

In style and structure, XML documents look quite similar to HTML documents. However, when a Web server with XML content prepares data for transmission, the Web server must generate a context wrapper with each XML fragment, including pointers to an associated Document Type Definition (DTD) and one or more style sheets for formatting. Clients for the Web server that process XML must be able to unpack the content fragment, parse the fragment in the context according to the DTD (if needed), render (if needed) in accordance with the specified style sheet guidelines, and correctly interpret the hypertext semantics (e.g. links) associated with each of the different document tags. It is understood that a DTD is not required for an XML document, instead, the author can simply use an application-specific tagset. However, a DTD is useful because it allows applications to validate the tagset for proper usage. The DTD specifies the set of required and optional elements and their attributes for documents to conform to that type. In addition, the DTD specifies the names of the tags and the

relationships among elements in a document, for example, nesting of elements.

One of the main issues during the development of XML was expressing data stored in XML documents into various formats. This has given rise to the development of languages and standards such as XSL and XSLT from W3C. By using an XSL style sheet one is able to create an HTML representation of an XML document. Similarly, XSL style sheets are used to transform XML documents into an HTML representation. XSL implementations suffer two principal limitations. First, XSLT requires the definition of rules to transform specific types of XML elements into some other type of XML like structure. Although the result may simply comprise a stream, the navigation rules must be defined based on types and the execution of the navigation rules is based on the degree of uniqueness of the navigation. In other words, the execution is not a simple procedural operation. Secondly, XSLT has no formatting capability which means that formatting is performed using flow/format objects in XSL and is geared towards HTML forming. Therefore, the XSL standards fall short for generating code or non-*ML output from a data model.

Another issue is concerned with the transformation of XML data. Frequently, XML data needs to be transformed into another valid XML format. This often comprises an update activity not a creation activity. The updates are made to specific pre-existing XML data, for example, converting selected strings into an alternate format, which is not the

same as creating a new document. To be effective, the transformation needs to include an efficient navigation mechanism.

Accordingly, there remains a need for a mechanism for navigating a data model and extracting specific data from the data model.

Summary of the Invention

The shortcomings of the prior art are overcome and additional advantages are provided through the provision of a system for updating a domain model and generating a formatted output. The system provides two principal functions: (a) text format generation from a source data model; and (b) manipulation of the DOM tree for the domain model.

The system provides the capability to navigate a source data model and extract specific data from the source data model to a target data model. The system utilizes a template driven mechanism and directives are provided to navigate the source data model and to change the flow of control of the template. The source data model includes read-only data that is extracted and used to generate the target data model. Directives are also provided to navigate the target data model.

Through the provision of a set of "structure walking directives" and a set of basic "string formatting"

directives, a user can write a procedural sequence of instructions to quickly extract data from a XML source data model (e.g. an XML document) and format the extracted data into a desired stream for a target data model. The target data model may comprise a simple ASCII file, or an HTML file, or generated content of some other file format. The mechanism according to the invention provides a target data model without restrictions.

According to another aspect of the invention, the mechanism, i.e. directives, is implemented utilizing XML thereby providing a language interface which is natural to most users.

In a first aspect, the present invention provides a mechanism for manipulating information in a source data model and creating a target data model, the mechanism includes, (a) a template module having a directive to manipulate selected data in the source data model; (b) a template processing module to process the directive contained in the template module; and (c) the template processing module further includes a component to generate a DOM tree for navigating the template module to manipulate said source data model.

In another aspect, the present invention provides a method for manipulating selected data in a source data model, the method comprises the steps of: (a) defining a template file having a directive specifying the data to be extracted from the source data model; (b) generating a DOM

tree for navigating the template file; and (c) navigating the template file and applying the directive to manipulate selected data in the source data model.

In yet another aspect, the present invention provides a computer program product for an application program for creating objects, the application program includes a utility for manipulating information in a source data model and creating a target data model, the computer program product comprises: a recording medium; means recorded on the medium for instructing a computer to perform the steps of, (a) defining a template file having a directive specifying the data to be extracted from the source data model; (b) generating a DOM tree for navigating the template file; and (c) navigating the template file and applying the directive to manipulate selected data in the source data model.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention.

Brief Description of the Drawings

The subject matter which is regarded as the invention is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the invention are apparent from the following detailed

description taken in conjunction with the accompanying drawings in which:

FIG. 1 shows in diagrammatic form a system for updating a domain model and generating a formatted output in accordance with the present invention;

FIG. 2 shows in flowchart form the method steps for processing directives according to the present invention;

FIG. 3 shows in flowchart form the method steps for executing directives according to the present invention;

FIG. 4 shows in flowchart form the method steps for resolving a macro according to the present invention;

FIG. 5 shows in flowchart form the method steps for resolving scoping according to the present invention;

FIG. 6 shows in diagrammatic form a formatted output generated according to the present invention; and

FIG. 7 shows in diagrammatic form a domain model manipulation example according to the present invention.

Best Mode for Carrying Out the Invention

Reference is first made to Fig. 1 which shows in diagrammatic form a mechanism according to the present invention for updating a domain model and generating a formatted output. The mechanism comprises a template-driven implementation. In the preferred embodiment, the mechanism is in an application development program for creating objects, etc., according to object-oriented principles and the domain models and trees define an object model.

The mechanism is indicated generally by reference 10 and comprises a template file or module 12 and a template driven emitter or template processing module 14. As will be described in more detail below, the template driven emitter 14 applies the template file 12 to a domain model 16 to produce a generated output file 18. In the context of the present invention, the domain model 16 comprises a source data model and the generated output file 18 comprises a target data model. The source data model 16 contains read-only data which is extracted by the mechanism 10 and used to generate a formatted output, i.e. the target data model 18. The target data model 18 contains both read/write data that is manipulated by the mechanism 10. According to the invention, the mechanism 10 provides the capability to navigate the data models 16, 18 and also manipulate the target data model 18.

The template file 12 is created as a text file with directives that navigate the source data file 16 and navigate and manipulate the target data file 18. While the mechanism 10 is described in the context of an XML based data model 16, the mechanism 10 is not concerned with the structure of the model 16. The template file 12 is created using a conventional editor. The mechanism 10 only responds to navigational, substitution and emit directives contained in the template files 12. As will be described in more detail below, the data model 16 is traversed in a symbolic fashion, i.e. naming the object relationships as strings, rather than direct object references. The attribute values are retrieved in the same fashion, i.e. by naming the attribute in a string.

In the preferred embodiment, the template file 12 is expressed in XML format and comprises directives and macros. The specification of the document type definition (DTD) for the template file 12 is provided below in Appendix I. The directives are commands or rules that define the flow of control. The macros provide for string manipulation. The mechanism 10 includes domain model navigation directives, domain model manipulation directives, output manipulation directives, logical operation directives, and code section directives. The domain model navigation directives are used to navigate the domain model (i.e. the source data model 16). The domain model manipulation directives are used to modify the domain model (i.e. target data model 18). The output manipulation directives are used to modify the generated output in the target data model 18. The logical

operation directives are used to change the logical processing of a template file 12. The code section directives are used to maintain code sections in the generated output (i.e. target data model 18). A full listing of the directives is provided below in Appendix II.

As described above, the source data model (i.e. domain model 16 in Fig. 1) contains read-only data that the mechanism 10 can extract and use to generate a formatted output (i.e. the target data model 18). A feature of the mechanism 10 according to the present invention is the ability to manipulate the target data model 18. According to this aspect of the invention, the target data model 18 contains read/write data which can be manipulated according to directives contained in the template file 12. A full listing of the directives is provided below in Appendix II, and directives for manipulating the target data model 18 include *updatetargetscope*, *targetscope*, *addtargetscope*, and *removetargetscope*.

The updatetargetscope directive is used to update an element of a DOM tree associated with a domain model. If the element does not exist it is inserted into the DOM tree. The syntax for the updatetargetscope directive is shown below in Appendix II. The following example describes the operation of the updatetargetscope directive:

Suppose the model defines an element "Class'' which has two other embedded elements. This document (i.e the source data file 16) is called "class.xml ".

```
class.xml
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template (i.e. template file 12):

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<define MACRO=newMethodName>addNew</define>
<targetscope NAME="Method">
<updatetargetscope NAME="name"
TYPE="ATTRIBUTE ">$newMethodName$</updatetargetscope>
</targetscope>
</_TDEBlock_>
```

The resulting DOM tree when the template is applied to the document "class.xml" is as follows:

```
<Class name="Set">
<Method name="addNew">
</Method>
<Method name="del">
</Method>
</Class>
```

It will be appreciated that the name of the first method changes from "add" to "addNew".

The targetscope directive is used to navigate to an element that is part of an existing DOM tree associated with. The syntax for the targetscope directive is described below in Appendix II. The following example illustrates the operation of the targetscope directive:

Suppose the model defines an element "Class'' which has two other embedded elements. This document (i.e. the source data file 16) is called "class.xml".

```
class.xml
<Classlist>
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>
```

Also consider the following template (i.e. the template file 12):

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
</targetscope>
```

```
</_TDEBlock_>
```

The template driven emitter 14 navigates to the Class element in the class.xml document. It will be understood that no emitted output is created since this directive just navigates the target data model 18 (i.e. the target document). In this case the targeted XML document is class.xml.

The addtargetscope directive is used to insert an element into a DOM tree associated with a domain model. The syntax for the addtargetscope directive is described below in Appendix II. The following example illustrates the operation of the addtargetscope directive.

Suppose the model defines an element ''Class'' which has two other embedded elements. This document (i.e. the source data file 16) is called "class.xml".

```
class.xml
<Classlist>
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>
```

Also consider the following template (i.e. the template file 12):

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
<addtargetscope NAME="Method">
<addtargetscope NAME="Argument"></addtargetscope>
</addtargetscope>
</targetscope>
</_TDEBlock_>
```

The resulting DOM tree when the template is applied to the document "class.xml" is as follows:

```
<ClassList>
<Class name="Set">
<Method>
<Argument/>
</Method>
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>
```

The removetargetscope directive is used to delete an element from a DOM tree associated with a domain model. The syntax for the removetargetscope directive is shown below in

Appendix II. The following example describes the operation of the removetargetscope directive:

Suppose the model defines an element "Class'' which has two other embedded elements. This document (i.e. the source data file 16) is called "class.xml".

```
class.xml
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template (i.e. the template file 12):

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<removetargetscope NAME="Method"/>
</_TDEBlock_>
```

The result of when the template is applied to the document "class.xml" is as follows:

```
<Class name="Set">
<Method name="del">
</Method>
</Class>
```

It will be appreciated that the first method element was removed. Since no index was specified the default value for the index is 0.

It will be appreciated that these directives are very useful in applications, such as "Web" applications and database applications, which require updating of a model stored in XML or a model stored in a database. The other directives included in the mechanism 10 according to the present invention are described below in Appendix II.

The mechanism 10 also includes XML specific directives such as *updatetargetdoctype*. These XML specific directives are for working with data models that are stored as XML (unlike other directives which are not XML data model specific).

The updatetargetdoctype directive is used to update the !DOCTYPE element of a DOM tree associated with a domain model. If the element does not exist it is inserted into the DOM tree. The syntax of the updatetargetdoctype is described in Appendix II below. The operation of the updatetargetdoctype is illustrated by the following example which involves adding a simple !DOCTYPE element (additional examples are provided in Appendix II).

Consider the following template file:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../../dtd/tde.dtd">
<_TDEBlock_>
```

```
<updatetargetdoctype PUBLIC_NAME="Chemistry"
PUBLIC_URL="http://sunsite.unc.edu/public/chemistry.dtd"
ROOT_ELEMENT_NAME="myroot"/>
</_TDEBlock_>
```

The result of when the template is applied is as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE myroot PUBLIC "Chemistry"
"http://sunsite.unc.edu/public/chemistry.dtd">
```

The mechanism 10 according to the present invention also includes macro transformations. A macro associates a name with a set of values and enables string substitution within the template file 12. The macro transformations are useful in situations that require text to be changed to uppercase, lowercase, truncated, etc. The following macro transformations are provided in the mechanism 10: *toupper*, *tolower*, *truncate*, *separate*, *strip*, *stripTrailing*, *stripLeading*, *change in_pattern out_pattern*, *numWords*, *words[i]*, and *word[i]*.

The toupper macro converts the macro value to all upper case letters. If an optional parameter 'i' (as described in Appendix II) is specified, then only the i'th character is converted to upper case. The tolower macro converts the macro value to all lower case letters. If the optional parameter 'i' is specified in that case, then only the i'th character is converted to lower case. The truncate macro truncates the macro value to a specified parameter 'maxlen'

characters long by removing vowels. If no maxlen parameter is specified, then the macro truncates to the default of 8 characters. The separate macro separates the values in a multivalue macro by separator. If the separator is a string/character, the separator must be enclosed by quotes ''. Otherwise it is assumed to be a macro name and the value of that macro is used as the separator. It is to be noted that when used in a <repeat> statement, repeat blocks are separated by a separator. The strip macro removes all leading and trailing white space characters from the macro value. If a string pattern is specified (either as a string or a macro name), the pattern will be stripped off from the macro value. The stripTrailing macro removes all trailing white space characters from the macro value. An optional pattern to strip may be specified. The stripLeading macro removes all leading white space characters from the macro value. An optional pattern to strip may be specified. The change in_pattern out_pattern macro replaces every occurrence of the in_pattern in the macro value with out_pattern. Strings/characters must be enclosed by quotes, otherwise the parameter is assumed to be a macro name and the value of that macro will be used. The increment macro increments the value of the macro by one. If an optional parameter 'i' is specified, then the increment step is defined by 'i'. The numWords macro returns the number of words in a string. For example, if the string contains "This is an apple pie", then the macro numWords returns '5'. The words[i] macro returns the string after the i'th word inclusively. For the previous example, words [3] would return "an apple pie". The word[i] macro returns the ith

word in a string. For the previous example, the macro word[3] would return "an". The macro transformations included in the mechanism 10 according to the present invention are further described in Appendix II.

According to another aspect, the mechanism 10 utilizes a "tree" navigation scheme to perform transformations in the source data model 16 (and the target data model 18). As will be described in more detail below, to apply a transformation to a node in the data model 16 or 18, the mechanism 10 first navigates the root node for the data model, then down to the child node until the specific node of interest is located. Once located, the transformation rule(s) is applied to the node. As tree navigation is not data model specific, the mechanism 10 according to the present invention is advantageously flexible and can support various data model types other than XML.

The mechanism 10 also includes a facility for code sections. The code sections are analogous to methods and have application for reusing templates, or for overriding other code sections with the same name. The code sections comprise the directives *code* and *call*, which are described in more detail in Appendix II.

Reference is next made to Figs. 2 to 5 which show the method steps embodied in processing the template file 12 according to the present invention.

Fig. 2 shows in flow chart form the method steps embodied in the mechanism 10 for processing directives according to the present invention. The method steps comprise a process which is indicated generally by reference 100 in Fig. 2. As described above, the directives for navigating and/or manipulating the data model are contained in the template file 12 (Fig. 1), and the template file 12 is processed by the template driven emitter 14 (Fig. 1). The first step as indicated by block 110 in Fig. 2 involves initializing the template driven emitter 14, which in the preferred embodiment is capable of processing XML documents and templates. Initialization of the template driven emitter 14 includes the following operations: creating and populating a macro table with predefined macros (which will be used for string substitution in the template file) and creating a code section table (which will be populated with code sections, defined by name using the code directive, that will be emitted on the occurrence of a call directive using the code sections' name(s)). Additionally, the template driven emitter generates a DOM tree for the template file and for the source data model, both trees used for navigating respectively the template file and the source data model. DOM or document object model is an API for HTML and XML documents established through the W3C Organization. DOM defines the logical structure of documents and the way a document is accessed and manipulated. See http://www.w3.org/dom for more information. As used herein, DOM trees are understood as used in the context of the DOM specification but need strictly follow the DOM specification. Indeed, as used herein, DOM trees could be

trees according to conventional understandings or defined according to other non-DOM specifications.

The next step (block 112) in the process 100 involves getting the first or next directive from the template file 12. As described above, the template file 12 is created as a text file with the directives specified according to the definitions contained in Appendix II. The next step in decision block 114 involves ascertaining whether the directive is a directive for the template driven emitter 14. (It will be appreciated that only template driven emitter directives can drive the operation of the template driven emitter 14.) If the directive retrieved in step 112 is not a template driven emitter (TDE) directive, then a check (decision block 128) is made to determine if there any further directives in the template file 12. If no, then the process 100 for processing the template file 12 is terminated. If yes, then the next directive in the template file 12 is retrieved (step 112) and a check is made to ascertain if the directive is a TDE directive (decision block 114).

Referring to Fig. 2, if the directive is a TDE directive (as determined in decision block 114), then the next step (block 116) in the process 100 involves parsing the content of the directive for any macros which may be included. If the content does include a macro value (decision block 118), then a process 300 for resolving the macro name is invoked in the block 120. The result of the macro name process 300 is a macro value for the macro name.

The macro name resolving process 300 is described below with reference to Fig. 4. As shown in Fig. 2, the next step in the process 100 involves determining if the macro value returned from the process 300 requires scoping (decision block 122). If the macro value requires scoping, then a process 400 for scoping the macro value is invoked in block 130. The macro value scoping process 400 is described in more detail below with reference to Fig. 5. Once the macro value has been scoped, the directive is executed as indicated by block 124 in Fig. 2. If the macro value does not require scoping as determined in decision block 122, then the next step is also executing the directive in block 124. If the directive does not contain a macro value as determined in decision block 118, then processing also proceeds to executing the directive in block 124. In block 124, the directive is executed 200 as will be described in more detail with reference to Fig. 3 below. The result of executing the directive, i.e. content, is written to an output buffer as indicated in block 126. Next, a check is made in decision block 128 to ascertain if there any more directives in the template file 12. If there are more directives in the template file 12, then the processing steps starting at block 112 are repeated as described above. If there are no more directives, then the processing of the template file 12 is completed and the process 100 ends.

The processing of the template file 12 (Fig. 1) is implemented in the template driven emitter 14 (Fig. 1) as will be apparent to one skilled in the art based on the foregoing description.

Reference is next made to Fig. 3, which shows the method steps embodied in a process for executing the directive as indicated by block 124 (Fig. 2). As shown in Fig. 3, the process for executing the directives is indicated generally by reference 200. The directives are processed according the definitions found in Appendix II. The first operation in the process 200 involves pushing the current directive context onto a stack as indicated in block 210. Next, the start tag of the directive is processed (block 212) by, among other things, parsing the directive and performing the steps required to execute the action of the directive in accordance with the relevant definitions found in Appendix II. It should be apparent to those skilled in the art how to implement the specifications and definitions of Appendix II. The next operation involves determining if the directive contains any 'children' in decision block 214. If there is a child, then the current context is set to the child directive in block 216, and pushed onto the stack in block 210. The steps in blocks 212 and 214 are repeated in order to navigate the DOM tree.

Referring to Fig. 3, if the directive does not contain a child or any more children (as determined in decision block 214), then the end tag of the directive is processed in block 218. The last operation in the directive processing involves popping the current directive context from stack (which was pushed in block 210) to restore the state data, as indicated in block 220. After this step, the directive processing is ended.

Reference is next made to Fig. 4 which shows the method steps embodied in a process for resolving the macro name as indicated by block 120 (Fig. 2). As shown in Fig. 4, the process for resolving the macro name is indicated generally by reference 300. The first step in resolving the macro name process 300 involves ascertaining if the macro name is stored in the macro table, as indicated by decision block 310. If the macro name is contained in the macro table, then the value for the macro name is extracted from the macro table, as indicated in block 312, and the process 300 is completed. If the macro name is not stored in the macro table (decision block 310), then a check is made to ascertain if the macro name is stored in the domain model, as indicated in decision block 314. If the macro name is stored in the model, then the value for the macro name is extracted from the model, as indicated in block 316, and the process 300 for resolving the macro name is completed. If the macro name is not stored in the model (as determined in decision block 314), then the value for the macro name is set to the name encased by a '$' character in the template file 12. After step 318, the process 300 is completed.

Reference is next made to Fig. 5, which shows the method steps embodied in a process for resolving the scoping of the macro name as indicated by block 130 (Fig. 2). As shown in Fig. 5, the process for resolving the scoping of the macro name, i.e. navigating the DOM tree, is indicated generally by reference 400. The first step in the process 400 for resolving the scope of the macro name involves

processing the root scope name in block 410. Next, a determination is made in decision block 412 to determine if the scope refers to a child node in the DOM tree for the model. If the scope is a child node, then the scope context is moved to the child node, as indicated in block 414. After step 414, a check is made in decision block 416, to determine if the scope name contains any additional scopes. If there are additional scopes, then the scope name is processed in block 410 and decision block 412 as described above.

Referring to Fig. 5, if the scope name does not refer to a child node (as determined in decision block 412), then a check is made in decision block 418 to ascertain if the scope name refers to a parent node in the DOM tree. If yes, then the scope context is moved to the parent node as indicated in block 420. After this step, a check is made to ascertain if the scope name contains additional scopes in decision block 416, as described above. If the scope name does not correspond to the parent node (decision block 418), then a check is made in decision block 422 to determine if the scope name refers to the root node in the DOM tree. If the scope name refers to the root node, then the scope context is moved to the root node, as indicated by block 424. Next, a check is made in decision block 416 to determine if the scope name contains additional scopes as described above.

The operation of the mechanism 10 according to the present invention is further described with reference to two examples depicted in Fig. 6 and Fig. 7, respectively.

Reference is made to Fig. 6, which depicts a formatted output file 28 generated from a source data file 26 according to a template file 22. The template file 22 is processed according to a template driven emitter 24. The arrangement of the template file 22 and the template driven emitter 24 are as described above. In operation, the template driven emitter 24 reads the source data file 26 and creates a DOM tree. The template driven emitter 24 also reads the template file 22 and creates a DOM tree (as described above). When the template file 22 is first processed, the template driven emitter 24 sets the current context to the root element in both DOM trees. Subsequently, navigation of the DOM tree is controlled by the scope directive(s) contained in the template file 22. As shown in Fig. 6, the template file 22 defines the root element as 'TDEBlock' (Line 3). In Line 4, the template file 22 includes a directive which defines the formatted output file 28 named by a macro value. Line 6 of the template file 22 includes a scope directive which changes the context (i.e. navigates the DOM tree) to the 'Method'. As shown in Fig. 6, the generated output in the output file 28 is indicated by 'void add()' in Line 2. The output file 28 is stored under the name 'Set.java', which was defined by the 'outfile' directive in the template file 22 (Line 4).

Reference is next made to Fig. 7, which shows an example of a DOM tree manipulation according to the present invention. As shown in Fig. 7, the template driven emitter 34 processes a template file 32 which modifies a source data file 36 to generate an output file 38. The output file 38 comprises the source file 36 associated with a modified DOM tree. As shown in Fig. 7, the template file 32 includes an 'addtargetscope' directive in Line 4 and another 'addtargetscope' directive in Line 5. As described above, the addtargetscope directive is used to insert an element into the DOM tree. For the template file 32, the first directive directs the addition of an element called 'Method' to the DOM tree. The second directive in the template file 32 directs the addition of an attribute 'name' to the element 'Method'. The element 'Method' with the attribute 'name' appears in Line 7 of the generated output file 38 as shown in Fig. 7. It will be appreciated that the data model is navigated in a symbolic manner, i.e. naming the object relationships as strings, rather than direct object references. Similarly, the attribute values are retrieved by naming the attribute string.

The detailed descriptions may have been presented in terms of program procedures executed on a computer or network of computers. These procedural descriptions and representations are the means used by those skilled in the art to most effectively convey the substance of their work to others skilled in the art. They may be implemented in hardware or software, or a combination of the two.

A procedure is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, objects, attributes or the like. It should be noted, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Further, the manipulations performed are often referred to in terms, such as adding or comparing, which are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary, or desirable in most cases, in any of the operations described herein which form part of the present invention; the operations are machine operations. Useful machines for performing the operations of the present invention include general purpose digital computers or similar devices.

Each step of the method may be executed on any general computer, such as a mainframe computer, personal computer or the like and pursuant to one or more, or a part of one or more, program modules or objects generated from any

programming language, such as C++, Java, Fortran or the like. And still further, each step, or a file or object or the like implementing each step, may be executed by special purpose hardware or a circuit module designed for that purpose.

In the case of diagrams depicted herein, they are provided by way of example. There may be variations to these diagrams or the steps (or operations) described herein without departing from the spirit of the invention. For instance, in certain cases, the steps may be performed in differing order, or steps may be added, deleted or modified. All of these variations are considered to comprise part of the present invention as recited in the appended claims.

Throughout the description and claims of this specification, the word "comprise" and variations of the word, such as "comprising" and "comprises", is not intended to exclude other additives, integers or processed steps.

While the preferred embodiment of this invention has been described in relation to the XML language, this invention need not be solely operate using the XML language. It will be apparent to those skilled in the art that the invention may equally be operable with other computer languages, such as SGML.

The invention is preferably implemented in a high level procedural or object-oriented programming language to

communicate with a computer. However, the invention can be implemented in assembly or machine language, if desired. In any case, the language may be a compiled or interpreted language.

While aspects of the invention relate to certain computer language and other technological specifications (e.g. the XML specification), it should be apparent that classes, objects, components, tags and other such software and technological items referenced herein need not fully conform to the specification(s) defined therefor but rather may meet only some of the specification requirements. Moreover, the classes, objects, components, tags and other such software and technological items referenced herein may be defined according to equivalent specification(s) other than as indicated herein that provides equivalent or similar functionality, constraints, etc. For example, instead of the XML language specification, tags and other such software and technological items referenced herein may be defined according to the SGML specification where applicable and appropriate.

The invention may be implemented as a mechanism or a computer program product comprising a recording medium. Such a mechanism or computer program product may include, but is not limited to, CD-ROMs, diskettes, tapes, hard drives, computer RAM or ROM and/or the electronic, magnetic, optical, biological or other similar embodiment of the program. Indeed, the mechanism or computer program product

may include any solid or fluid transmission medium, magnetic or optical, or the like, for storing or transmitting signals readable by a machine for controlling the operation of a general or special purpose programmable computer according to the method of the invention and/or to structure its components in accordance with a system of the invention.

The invention may also be implemented in a system. A system may comprise a computer that includes a processor and a memory device and optionally, a storage device, an output device such as a video display and/or an input device such as a keyboard or computer mouse. Moreover, a system may comprise an interconnected network of computers. Computers may equally be in stand-alone form (such as the traditional desktop personal computer) or integrated into another apparatus (such a cellular telephone). The system may be specially constructed for the required purposes to perform, for example, the method steps of the invention or it may comprise one or more general purpose computers as selectively activated or reconfigured by a computer program in accordance with the teachings herein stored in the computer(s). The procedures presented herein are not inherently related to a particular computer system or other apparatus. The required structure for a variety of these systems will appear from the description given.

While this invention has been described in relation to preferred embodiments, it will be understood by those skilled in the art that changes in the details of

construction, arrangement of parts, compositions, processes, structures and materials selection may be made without departing from the spirit and scope of this invention. Many modifications and variations are possible in light of the above teaching. Thus, it should be understood that the above described embodiments have been provided by way of example rather than as a limitation and that the specification and drawing(s) are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

The present invention can be included in an article of manufacture (e.g., one or more computer program products) having, for instance, computer usable media. The media has embodied therein, for instance, computer readable program code means for providing and facilitating the capabilities of the present invention. The article of manufacture can be included as a part of a computer system or sold separately.

Additionally, at least one program storage device readable by a machine, tangibly embodying at least one program of instructions executable by the machine to perform the capabilities of the present invention can be provided.

The flow diagrams depicted herein are just examples. There may be many variations to these diagrams or the steps (or operations) described therein without departing from the spirit of the invention. For instance, the steps may be performed in a differing order, or steps may be added, deleted or modified. All of these variations are considered a part of the claimed invention.

Although preferred embodiments have been depicted and described in detail herein, it will be apparent to those skilled in the relevant art that various modifications, additions, substitutions and the like can be made without departing from the spirit of the invention and these are therefore considered to be within the scope of the invention as defined in the following claims.

APPENDIX I

DTD Specifications

```
<?xml encoding="US-ASCII"?>
<!ENTITY % tdechildren "(#PCDATA | outfile | define | clear
| if | repeat |
scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
ifhasrepeatscope | hasscope | targetscope )*" >
<!ENTITY % targettdechildren "(#PCDATA | outfile | define |
clear | if | repeat | scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
repeattargetscope |
ifhasrepeatscope | hasscope | updatetargetscope |
addtargetscope |
removetargetscope | ifhastargetscope |
ifhasrepeattargetscope | hastargetscope |
targetscope | updatetargetdoctype )*" >
<!ENTITY % ifhastdechildren "(#PCDATA | outfile | define |
clear | if | else |
repeat | scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
repeattargetscope |
ifhasrepeatscope | hasscope | updatetargetscope |
addtargetscope |
removetargetscope | ifhastargetscope |
ifhasrepeattargetscope | hastargetscope |
targetscope | updatetargetdoctype )*" >
```

```
<!ENTITY % elsetdechildren "(#PCDATA | outfile | define |
clear | if | repeat |
scope | repeatscope | createalias |
include | code | call | sameline | assert | ifhasscope |
ifhasrepeatscope |
hasscope | else | targetscope )*" >
<!ELEMENT define ( #PCDATA ) >
<!ATTLIST define MACRO CDATA #REQUIRED>
<!ELEMENT clear EMPTY >
<!ATTLIST clear MACRO CDATA #REQUIRED>
<!ELEMENT _TDEBlock_ %targettdechildren; >
<!ATTLIST _TDEBlock_ DOMTree CDATA #IMPLIED>
<!ELEMENT updatetargetdoctype EMPTY >
<!ATTLIST updatetargetdoctype SYSTEM_URL CDATA #IMPLIED>
<!ATTLIST updatetargetdoctype PUBLIC_URL CDATA #IMPLIED>
<!ATTLIST updatetargetdoctype PUBLIC_NAME CDATA #IMPLIED>
<!ATTLIST updatetargetdoctype ROOT_ELEMENT_NAME CDATA
#IMPLIED>
<!ELEMENT if %elsetdechildren; >
<!ATTLIST if EXPRESSION CDATA #REQUIRED>
<!ELEMENT else EMPTY >
<!ELEMENT addtargetscope %targettdechildren; >
<!ATTLIST addtargetscope NAME CDATA #REQUIRED>
<!ATTLIST addtargetscope TYPE (ELEMENT | ATTRIBUTE)
"ELEMENT">
<!ATTLIST addtargetscope INDEX CDATA #IMPLIED >
<!ELEMENT updatetargetscope %targettdechildren; >
<!ATTLIST updatetargetscope NAME CDATA #REQUIRED>
<!ATTLIST updatetargetscope TYPE (ELEMENT | ATTRIBUTE)
"ELEMENT">
```

```
<!ATTLIST updatetargetscope INDEX CDATA #IMPLIED >
<!ELEMENT repeattargetscope %targettdechildren; >
<!ATTLIST repeattargetscope NAME CDATA #REQUIRED>
<!ELEMENT removetargetscope EMPTY >
<!ATTLIST removetargetscope NAME CDATA #REQUIRED>
<!ATTLIST removetargetscope TYPE (ELEMENT | ATTRIBUTE)
"ELEMENT">
<!ATTLIST removetargetscope INDEX CDATA #IMPLIED >
<!ELEMENT targetscope %targettdechildren; >
<!ATTLIST targetscope NAME CDATA #REQUIRED>
<!ATTLIST targetscope INDEX CDATA #IMPLIED >
<!ELEMENT hastargetscope %targettdechildren; >
<!ATTLIST hastargetscope NAME CDATA #REQUIRED>
<!ATTLIST hastargetscope INDEX CDATA #IMPLIED >
<!ELEMENT ifhastargetscope %ifhastdechildren; >
<!ATTLIST ifhastargetscope NAME CDATA #REQUIRED>
<!ATTLIST ifhastargetscope INDEX CDATA #IMPLIED >
<!ELEMENT ifhasrepeattargetscope %ifhastdechildren; >
<!ATTLIST ifhasrepeattargetscope NAME CDATA #REQUIRED>
<!ELEMENT repeat %tdechildren; >
<!ATTLIST repeat MACRO CDATA #REQUIRED>
<!ELEMENT repeatscope %tdechildren; >
<!ATTLIST repeatscope NAME CDATA #REQUIRED>
<!ELEMENT scope %targettdechildren; >
<!ATTLIST scope NAME CDATA #REQUIRED>
<!ELEMENT include EMPTY >
<!ATTLIST include FILENAME CDATA #REQUIRED>
<!ELEMENT outfile %tdechildren; >
<!ATTLIST outfile FILENAME CDATA #REQUIRED>
<!ATTLIST outfile MODE CDATA #IMPLIED>
```

```
<!ELEMENT code %tdechildren; >
<!ATTLIST code NAME CDATA #REQUIRED>
<!ELEMENT call %tdechildren; >
<!ATTLIST call NAME CDATA #REQUIRED>
<!ATTLIST call MACROLIST CDATA #IMPLIED>
<!ELEMENT sameline %tdechildren; >
<!ELEMENT assert EMPTY >
<!ATTLIST assert EXPRESSION CDATA #REQUIRED>
<!ELEMENT ifhasrepeatscope %ifhastdechildren; >
<!ATTLIST ifhasrepeatscope NAME CDATA #REQUIRED>
<!ELEMENT hasscope %tdechildren; >
<!ATTLIST hasscope NAME CDATA #REQUIRED>
<!ELEMENT ifhasscope %ifhastdechildren; >
<!ATTLIST ifhasscope NAME CDATA #REQUIRED>
<!ELEMENT createalias %targettdechildren; >
<!ATTLIST createalias ALIASNAME CDATA #REQUIRED>
<!ATTLIST createalias ALIASPATH CDATA #REQUIRED>
```

APPENDIX II

Background

The preferred embodiment of the mechanism according to the present invention for updating a domain model and generating a formatted output works with an XML model, however, the mechanism is unaware of the structure of the model. It only responds to navigational, substitution and emit directives contained in the templates. The model is traversed in a symbolic fashion (i.e. naming the object relationships as strings, rather than direct object references), and the attribute values are retrieved in the same fashion (i.e. by naming the attribute in a string).

Language Specification

Introduction

For ease of use, the template language is a simple language that has a well formed XML syntax. The language consists of the following constructs:

1. Text. Text is emitted as is. Blanks are treated as text; any indentation that occurs in the template file will be echoed in the output.

2. Macros. Macros associate a name with a set of values. They enable string substitution within templates.

3. Directives. Directives are commands that control code generation.

4. Comments. Comments help to clarify the template and do not get emitted into the output file.

Throughout this Appendix there are references to the '[' and ']' character. These characters emphasize that the parameter within these characters are optional. For example consider the following:

transform = toupper [index]

wherein the 'index' parameter is optional.

Template Files

Template files are flat text files that are portable across different platforms. They can be edited with any text editor. White space is treated as-is and new line characters are expected at the end of each line. In the preferred embodiment, the template files conform to the XML specifications on each platform.

Macros

Macros enable string substitution within templates.

Macros Syntax

$macro_name$

Description

$macro_name$ defines a macro name that will be substituted with its assigned string value. The macro instance is replaced with the macro value as the output is emitted. A

macro name consists of lower and upper case letters, and digits.

A macro is an implicit array: a simple macro has either one string value, or a (ordered) sequence of string values; a macro structure is either a structure of macro fields (which, in turn, are implicit arrays) or it is an array of identical structures. Here identical structures is meant to imply that all the fields of the structures have the same names.

Example 1 (simple macro)
Consider the following template definition:

```
..
class $CLASS_NAME$
{
};
..
```

If the user defines the macro CLASS_NAME to be GuiObject, then the following will be emitted:

```
class GuiObject
{
};
```

Macro Transformations

Macro transformations allow you to take a macro value, convert it to some other form, and then use the result of the transformation as the macro value. A transformation can be used whenever a macro is used.

Syntax

$macro_name| transform$

The first transformation type (|) does not permanently change the value of the macro, but only during that particular macro emit. The second transformation type (=) permanently changes the value of the macro.

The mechanism recognizes the following transformations: (note: [parameter] means optional parameter and the user must provide an index when using "[ ]")

```
transform = toupper [i] |
            tolower [i] |
            truncate [maxlen] |
            separate separator |
            strip [pattern] |
            stripTrailing [pattern] |
            stripLeading  [pattern] |
            change in_pattern out_pattern |
            increment [i] |
            numWords |
            words [i] |
            word[i]
```

Description

The following describes each transform:

toupper - converts the macro value to all upper case letters. If the optional parameter i is specified, then only the i'th character is converted to upper case.

tolower - converts the macro value to all lower case letters. If the optional parameter i is specified, then only the i'th character is converted to upper case.

truncate - truncates the macro value to maxlen characters long by removing vowels. With no parameter, the default is to truncate to 8 characters.

separate - separates the values in a multivalue macro by a separator. If the separator is a string/character, it must be enclosed by quotes ''. Otherwise it is assumed to be a macro name and the value of that macro is used as a separator. When used in a <repeat> statement, repeat blocks are separated by a separator.

strip - removes all leading and trailing white space characters from the macro value. If a string pattern is specified (either as a string or a macro name), the pattern will be stripped off from the macro value.

stripTrailing - removes all trailing white space characters from the macro value. An optional pattern to strip can be specified

stripLeading - removes all leading white space characters from the macro value. An optional pattern to strip can be specified.

change in_pattern out_pattern - replaces every occurrence of the in_pattern in the macro value with the out_pattern. Again, strings/characters must be enclosed by quotes, else the parameter is assumed to be a macro name and the value of that macro will be used.

increment - increments the value of the macro by one. If the optional parameter i is specified, then the increment step is i.

numWords - returns the number of words of a string. If the string contains "This is an apple pie", numWords returns 5.

words[i] - returns the string after the ith word inclusively. In the previous example, words [3] would return "an apple pie".

word[i] - returns the ith word in a string. In the previous example, word[3] would return "an".

Example 1:
Consider the following template definition:

```
#define _$CLASS_NAME|toupper$_HPP_
```

```
/*
 *  File name: $CLASS_NAME|truncate$
 */
```

If the user defines the macro CLASS_NAME to be GuiObject, then the following will be emitted:

```
#define _GUIOBJECT_HPP_
/*
 *  File name: GiObject
 */
```

Example 2:
Consider the following template:

```
<_TDEBlock_>
<define MACRO="FILE_NAME">myFile.hpp</define>
<define MACRO="EXTENSION">hpp</define>
 $FILE_NAME| change EXTENSION ''cpp''$
</_TDEBlock_>
```

The above template file defines a macro called FILE_NAME whose string value is myFile.hpp and EXTENSION is a macro whose string value is hpp. For information regarding the define directive, refer below. The resulting output is emitted:

myFile.cpp

Example 3:
Consider the following template:

```
<_TDEBlock_>
<define MACRO="firstname">John</define>
<define MACRO="lastname">Doe</define>
 $firstname= toupper$ $lastname= toupper$
 $firstname$ $lastname$
</_TDEBlock_>
```

In this case, the "=" transformation type is used instead of the "|" transformation type. Note that this template file defines a firstname and lastname macro. The values of the firstname and the lastname macros are uppercased. Once this is done the values of the firstname and lastname macros are always uppercased. For information regarding the define directive, refer below. The resulting output is emitted:

```
JOHN DOE
JOHN DOE
```

Note that the values of firstname and lastname macros retain their transformation (ie the value of each macro is uppercased).

Directives

Directives define the flow of control in the template files. Directives are enclosed within ''< >'' delimiters (e.g. <name>) and can appear anywhere in the template. XML imposes some restrictions on attribute values. For example to form a well-formed XML document the characters '<' and '&' may only be used to start tags and entities. Therefore,

entity references must be used to represent the '<' and '&' characters. The following is list of entity references:

Entity Reference	Character Representation
&amp;	&
&lt;	<
&gt;	>
&apos;	'
&qout;	"

Certain directives must be used in begin/end pairs using the notation <name> ... </name>.

A directive with nothing inside is an empty directive and has no effect in the emitted text. If a directive starts in the middle of a line, the leading white space is taken as indentation for the language and it will not be emitted.

The following directives are supported:

TDEBlock
define
clear
if
repeat
scope
repeatscope
include
outfile
code
call

sameline
assert
hasscope
ifhasscope
ifhasrepeatscope
targetscope
repeattargetscope
ifhasrepeattargetscope
addtargetscope
removetargetscope
updatetargetscope
updatetargetdoctype
addattribute
updateattribute
removeattribute
addtext
updatetext
removetext
hastargetscope
ifhastargetscope
createalias

TDEBlock Directive

TDEBlock is the only required directive that should be the root element for each template file.

Syntax

<_TDEBlock_ SourceDOMTree="sourcedoc.xml" TargetDOMTree="targetdoc.xml">

section
</_TDEBlock_>

Description
A source and target XML document can be specified in this directive. By specifying a target XML document one can write directives to modify the targeted XML document. By specifying a source XML document one can read data from the XML document. For more information see the "Modifying Existing DOM Trees" section.

Example 1: Simple template file
<_TDEBlock_>
</_TDEBlock_>

Example 2: Simple template file that specifies a target XML document
<_TDEBlock_ TargetDOMTree="doc.xml">
</_TDEBlock_>

Example 3: Simple template file that specifies a target XML document and a source XML document.
<_TDEBlock_ SourceDOMTree="sourcedoc.xml"
TargetDOMTree="doc.xml">
</_TDEBlock_>

define Directive

The define directive associates the specified value, macro_value, with the macro_name. NOTE: <define> can only be used to define simple macros. \

Syntax

<define MACRO="macro_name [| transformation]">value</define>

Description

The define directive takes a single macro_value and assigns it to macro_name. The macro_value will be interpreted as is. If a transformation is present, macro_value is transformed before being assigned to macro_name. Multiple values can be assigned to macro_name by performing subsequent definitions. For macros that have multiple value assignments, the values are concatenated together (without separators) and treated as a single string. See below for multi-valued macros usage in a repeat directive. The macro definition can be removed by the clear directive.

NOTE:

<define MACRO="MACRO">value</define>

is different from

<define MACRO="MACRO">
value
</define>

as an extra newline character is added by the later.

Example 1: Single-value macro

Consider the following template definition:

<_TDEBlock_>

```
<define MACRO="String">
Hi there. \
How are you?
</define>
$String$
</_TDEBlock_>
```

The resulting output emitted is:

```
 Hi there. How are you?
```

The backslash at the end of the first line prevents the end of line character from being embedded in the string. If `\' is omitted, the text will be emitted on two lines as is.

Example 2: Multi-value macro

Consider the following template definition:

```
<_TDEBlock_>
<define MACRO="T_Class">Parent_</define>
<define MACRO="T_Class">Class</define>
<define MACRO="M_Class">my_class</define>
class $M_Class$ : public $T_Class$
{
   ARG
};
</_TDEBlock_>
```

When you run the TDE, the following is emitted:

```
 class my_class : public Parent_Class
```

```
{
   ARG
};
```

Example 3: Transformation applied to macros

Consider the following template definition:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../tde.dtd">
<_TDEBlock_>
    <outfile FILENAME="define.log">
    <define MACRO="macro1|toupper|strip E">one</define>
    <define MACRO="macro2|tolower|strip E">one</define>
    <define MACRO="macro3|tolower|strip e">ONE</define>
    $macro1$
    $macro2$
    $macro3$
    </outfile>
<_TDEBlock_>
```

The following is emitted to the "define.log" file:

```
    ON
    one
    on
```

The above template file defines a macro called "macro1" that goes through two transforms. The first transform uppercases the value of this macro, in this case the value changes from "one" to "ONE". After that transformation, the "E" is stripped from the end. The resulting macro is "ON". The

second macro also goes through two transforms. The first transform lowercases the macro value from "one" to "one". In this case though this transform is meaningless since the macro value is already lower. After the tolower transform, an "E" character is stripped from the end. Note that this transform will fail since the "strip" transform is looking for a capital "E" character and the value of the macro is all lowercased. Therefore, the resulting macro value is "one". The last macro - macro3 - also goes through two transforms. The first transform lowercases the macro value from "ONE" to "one". The second transform strips the "e" character from the macro value. Note that at this point the macro value is all lowercased, therefore the resulting macro value will be "on".

clear Directive

The clear directive is used to clear the value for the macro associated with the macro_name supplied.

Syntax

<clear MACRO="macro-list">

where,

macro-list = macro_name [, macro-list]

Description

This directive is useful for changing the value of a macro_name, rather than adding to it (as the define directive does). Using a macro_name that has been previously cleared has the same effect as using a macro_name that has never been defined. Furthermore,

issuing a clear on a macro which has not been defined has no effect.

Example
The following clears the macro MEMBER_NAME.

```
<_TDEBlock_>
<define MACRO="MEMBER_NAME">foo</define>
<clear MACRO="MEMBER_NAME">
$MEMBER_NAME$
</_TDEBlock_>
```

The resulting emitted output is as follows:

```
$MEMBER_NAME$
```

The macro MEMBER_NAME has a value of "foo" then it is cleared. When the macro is cleared it is as though the macro was never defined. Therefore the generated output would print out the macro name instead of the value (i.e. $MEMBER_NAME$).

if Directive
The if directive facilitates the emission of a section of code based on some condition(s). The syntax used for filtering is similar to that used for conditional branches in C.

Syntax

```
<if EXPRESSION="expr">
<else/>
</if>
```

where,

expr= (expr logical-operator expr)
(value cond value)
($macro_name$)
(!expr)
(!$macro_name$)

logical-operator = & | |
cond = == | != | < | > | <= | >=
value = string | $macro_name$

Description

The conditional expression must be fully parenthesized. An expression takes the form of a value, followed by a condition, then followed by a value. The value can be a string or another macro_name. The expression evaluates to a string comparison. The else clause is optional. If the expression is too long, you can use the ''\'' as an escape character to specify it on multiple lines. The ! is used as a negation operator. If strings contain blanks they can be enclosed by quotes. The `<` and `>' operators are used for string containment: A < B evaluates to true if A is a substring of B (e.g. S1 < S2 evaluates if S1 is a substring of S2.) On the other hand, '<=' and '>=' have a different interpretation compared to '<' and '>'. '<=' and '>=' are relational operators used for comparing expressions. (e.g (value <=1) evaluates if value is smaller or equal to 1.)

Example #1

This example illustrates the use of nested ifs:

<_TDEBlock_>

```
    <if EXPRESSION="($METHOD_NAME$ == &qout;queue&qout;)">
    code to emit if method name is ''queue''
    <else/><if EXPRESSION="($METHOD_NAME$ ==
&qout;print&qout;)">
         code to emit if method name is ''print''
    <else/>
         code to emit otherwise...
    </if></if>
</_TDEBlock_>

Example #2
This example illustrates the use of the less-than (<)
operation and the and (&) operation. The conditional
operators act on string values.

<_TDEBlock_>
<outfile FILENAME="if.log">
<define MACRO="TotalMethods"> print delete add </define>
<define MACRO="MY_METHOD">print</define>
<define MACRO="DONE">no</define>
<if EXPRESSION="( (!($MY_METHOD$ &lt; $TotalMethods$)) &amp;
( $DONE$ != &qout;yes&qout;))">
      code to emit if we're not done and my method is
unknown
</if>
</outfile>
</_TDEBlock_>

The emitted code is as follows:
```

```
        code to emit if we're not done and my method is
unknown
```

Since the MY_METHOD macro is a substring of TotalMethods and the DONE macro is not equal to "yes" the section of text within the if statement is emitted.

Example #3

This example illustrates the use of a single macro_name:

```
<_TDEBlock_>
<if EXPRESSION="($MEMBER$)">
    code to emit if member has a value
</if>
</_TDEBlock_>
```

No code is emitted since the MEMBER macro is not defined.

repeat Directive

The repeat directive allows a single section of a template to emit over and over again on multivalue macro. It acts as a macro iterator.

Syntax

```
<repeat MACRO="macro-list [| transformation]">
section
</repeat>
```

where

macro-list = macro_name [,macro-list]

Description

The repeat directive will emit a section, within the start and end delimiters, repeatedly for each value associated with macro_name. The macro-list defines which macros to iterate over. If a macro has multiple values, each value will be emitted once in the repeat section. In essence, the repeat is analogous to a for loop, looping through all the macro values. If a transformation is present, it applies to the text contained within the repeat section. The number of iterations is the smallest number of values the macros have. If one of the macros is undefined, the code inside the <repeat> ... </repeat> will not be emitted.

Example 1
Consider the following template definition:

```
<_TDEBlock_>
        <define MACRO="COUNT">1</define>
        <define MACRO="COUNT">2</define>
        <define MACRO="COUNT">3</define>
        <repeat MACRO="COUNT">
        list.add(person$COUNT$);
        </repeat>
</_TDEBlock_>
```

When the template is parsed, the following is emitted:

```
        list.add(person1);
        list.add(person2);
        list.add(person3);
```

Example 2
Consider the following template definition:

```
<_TDEBlock_>
        <define MACRO="TYPE">int</define>
        <define MACRO="TYPE">char</define>
        <define MACRO="FUNC">print</define>
        <define MACRO="FUNC">display</define>
        <repeat MACRO="TYPE,FUNC">
        void $FUNC$ ($TYPE$ x);
        </repeat>
</_TDEBlock_>
```

When the template is parsed, the following is emitted:

```
        void print (int x);
        void display (char x);
```

Example 3

Consider the following template definition

```
 <_TDEBlock_>
       <define MACRO="TYPE">int</define>
        <define MACRO="TYPE">char</define>
        <define MACRO="PARAM">i</define>
        <define MACRO="PARAM">c</define>
void func(
<repeat MACRO="TYPE,PARAM | separate ',' " >
        $TYPE$ $PARAM$
</repeat>
        );
</_TDEBlock_>
```

When the template is parsed, the following is emitted:

```
void func(
        int i,
        char c
        );
```

scope Directive

The scope directive is used to navigate the XML model and specify how the macros in a section are scoped.

Syntax

```
 <scope NAME="sname">
section
</scope>
```

Description

The sname specifies the name of the scope.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements.

```
<Class name="Set">
<Method name="add">
</Method>
</Class>
```

When the element ''Class'' is generated, using the following template:

```
<_TDEBlock_>
```

```
class $name$ {            <-- this $name$ is from the Class
object
  <scope NAME="Method">
        void $name$();
  </scope>
};
</_TDEBlock_>
```

this will be emitted as:

```
class Set {
        void add();
}
```

In order to make scoping conceptually easy to use, a 'directory' navigation approach has been adopted. By specifying a '//' an implicit scope operation is implied. For example, consider the following XML document.

```
<classlist>
   <class classname="MyClass">
      <datamember>Member1</datamember>
   </class>
</classlist>
```

And the following template file:

```
<_TDEBlock_>
   <scope NAME="class">
      <scope NAME="datamember">
```

```
      </scope>
   </scope>
</_TDEBlock_>
```

The above template file can also be written as:

```
<_TDEBlock_>
<scope NAME="class//datamember">
</scope>
</_TDEBlock_>
```

It is also possible to write a template without using the scope directive. Consider the following XML document.

```
<Class name="Set">
<Method name="add">
</Method>
</Class>
```

Also consider the following template file.

```
<_TDEBlock_>
class $name$ {            <-- this $name$ is from the Class
object
        void $Method//name$();
};
</_TDEBlock_>
```

The following is the resulting emitted code:

```
class Set {
```

```
        void add();
};
```

The scoping navigation starts at the current scope. However, navigation can occur relative to the root element by specifying a '//' in front of the scoping pattern. Parent navigation can also be achieved by specifying '..' within the scoping pattern. Consider the following XML document.

```
<classlist>
   <class classname="MyClass">
      <datamember>Member1</datamember>
   </class>
</classlist>
```

And, the following template file.

```
<_TDEBlock_>
<scope NAME="class">
      <scope NAME="datamember">
            <scope NAME="..">
                  $classname$
            </scope>
      </scope>
</scope>
</_TDEBlock_>
```

The following would be emitted:

```
MyClass
```

An XML document can consists of elements of the same type. In order to navigate elements of the same type an index is specified to represent a specific element. For example the following XML document consists of three elements of the same type.

```
<classlist>
   <class>
       <method>init</method>
       <method>read</method>
       <method>write</method>
   </class>
</classlist>
```

Scoping can occur based on the position of the element. This is illustrated in the following template file.

```
<_TDEBlock_>
<scope NAME="class">
     <scope NAME="method[1]">
            $TEXT$
     </scope>
     <scope NAME="method[2]">
            $TEXT$
     </scope>
     <scope NAME="method[0]">
            $TEXT$
     </scope>
</scope>
```

</_TDEBlock_>

The following would be emitted:

read
write
init

repeatscope Directive

The repeatscope directive allows a single section of a template to emit over and over again on a list of elements. It acts as an element iterator.

Syntax

<repeatscope NAME="sname" SEPARATOR="separator">
section
</repeatscope>

Description

The sname specifies the name of the scope. The separator specifies the delimiter to use between sections. The default value is an empty string.

Example 1

Suppose the model defines an element ''Class'' which has two other embedded elements.

<Classlist>
<Class name="Set">
<Method name="add">
</Method>

```
<Method name="del">
</Method>
</Class>
</Classlist>
```

When the element ''Class'' is generated, using the following template:

```
<_TDEBlock_>
<scope NAME="Class">
   <repeatscope NAME="Method">
        $name$
   </repeatscope>
</scope>
</_TDEBlock_>
```

The following would be emitted.

```
        add
        del
```

Example 2

Consider the following XML document. This example shows how one can use the 'SEPARATOR' attribute.

```
<Elementlist>
<Element name="A"/>
<Element name="B"/>
<Element name="C"/>
<Element name="D"/>
<Element name="E"/>
```

CA919990037US1

```
<Element name="F"/>
<Element name="G"/>
</Elementlist>
```

And, the following template file.

```
<_TDEBlock_>
   <repeatscope NAME="Element" SEPARATOR=",">
$name$
   </repeatscope>
</_TDEBlock_>
```

The resulting emitted code is as follows:

```
A,B,C,D,E,F,G
```

include directive

The include directive imbeds a single input template file in-line.

Syntax

```
<include FILENAME="input_file_name"/>
```

Description

The include directive includes an entire file specified by input_file_name, into the input stream for processing. It is analogous to the #include file_name C directive. The input_file_name must be an input template file.

Example

A file called comment.tde contains the following:

```
//------------------------------
// C++ Generated Code
//------------------------------
```

Consider the following template file which includes comment.tde:

```
<_TDEBlock_>
<include FILENAME="comment.tde"/>
          class A {};
</_TDEBlock_>
```

The following is emitted:

```
//------------------------------
// C++ Generated Code
//------------------------------
class A {};
```

outfile Directive

The outfile directive specifies the output file to write to for that section of the template.

Syntax

```
<outfile FILENAME="outfile_name" MODE="_mode_">
section
</outfile>
```

where,

outfile_name = $macro_name$ | string

mode = append | new

Description

When the outfile directive is encountered, the specified outfile_name is opened with the specified mode, and the section of the template is written to that output file.

Example

Consider the following template definition:

```
<_TDEBlock_>
<outfile FILENAME="$filename$" MODE="new">
// C++ generated code
foo();
<outfile FILENAME="d:\\log.dat" MODE="append">     <-- need
to escape the backslash
This is a log entry.
</outfile>
// End C++ generated code
</outfile>
</_TDEBlock_>
```

If the macro filename is defined to be foo.cpp, the following is emitted to foo.cpp:

```
// C++ generated code
foo();
// End C++ generated code
```

The file d:\log.dat will contain the following:

```
This is a log entry.
```

code Directive

The code directive identifies a section of code to emit when the cname value is set.

Syntax

```
<code NAME="cname">
section
</code>
```

A code section can also be declared with parameters, in which case the call directive must be used to emit it.

```
<code NAME ="cname, param1, param2, ..., param_n">
section
</code>
```

Description

The code directive takes a single cname and emits the delimited section if cname is invoked. If cname is not set, the section is not emitted. Code sections are useful for reusing templates, or for overriding other code sections with the same name. When using the <scope> directive, code sections can also be invoked recursively. The terminating condition is implicitly defined by the model, that is, when the code section is invoked from within a <scope> section that will eventually not be reached.

Example1

Consider the following template:

```
<_TDEBlock_>
 <code NAME ="C_FORM">
  malloc(sizeof(Fred));
  </code>
  <code NAME ="CPP_FORM">
  Fred* pTemp = new Fred;
  </code>
  <call NAME ="CPP_FORM"/>
  return ok;
</_TDEBlock_>
```

The following is emitted:

```
    Fred* pTemp = new Fred;
    return ok;
```

call directive

The call directive is the only way of invoking code sections. (Recall that a code section can also be emitted as a macro, by enclosing its name with $. This only works for code sections with no parameters).

Syntax:

<call NAME="codeName" MACROLIST="macro1 macro2 ... macro_n"/>

All the arguments of the call directive (a code name followed by macro names) can be separated by commas.

Example:

```
<_TDEBlock_>
<code NAME="C, M, N">
   M = $M$
   <repeat MACRO="N">
       N = $N$
   </repeat>
</code>
<call NAME="C" MACROLIST="M, N"/>
</_TDEBlock_>
```

If macro M was defined to have value ''foo'' and N to have two values, ''bar'' and ''BAR'' then the following is emitted:

```
   M = foo
       N = bar
       N = BAR
```

sameline Directive

Syntax

```
<sameline>
section
</sameline>
```

Description

The sameline directive is used to put its contained section on a single output line. Basically, all the newline characters from the section are removed and what follows after </sameline> will start on a new line.

Example1
Consider the following template:

```
 <_TDEBlock_>
  <sameline>
   int func(
   <repeat MACRO="PARMS | separate ',,'">
   $PARMS$
   </repeat>
   );
   </sameline>
</_TDEBlock_>
```

When the template is parsed with PARMS containing values ''int a'' and ''char c'', the following is emitted:

```
    int func(int a, char c);
```

assert Directive

Syntax

```
<assert EXPRESSION="expr"/>
expr= (expr logical-operator expr)
            (value cond value)
            ($macro_name$)
            (!expr)
            (!$macro_name$)
logical-operator = & | |
```

```
cond = == | != | < | > | <= | >=
value = string
```

Description

The assertion directive is used to test a boolean expression. If this expression is evaluated to false, an assertion failure exception is thrown.

Example1

Consider the following template:

```
<_TDEBlock_>
   <define MACRO="macroA">Bird</define>
   <define MACRO="macroB">Bee</define>
   <sameline>
   int func(
   <assert EXPRESSION="$macroA$ &lt; $macroB$"/>
   );
   </sameline>
</_TDEBlock_>
```

When the template is parsed nothing is emitted since Bird is not a substring of Bee. Instead an assertion failure exception is thrown.

hasscope directive

The hasscope directive is used to determine if a scope exists within the XML model.

Syntax

```
<hasscope NAME="sname">
```

section
</hasscope>

Description
The sname specifies the name of the scope. This directive does not navigate the XML model. If the scope name does exist the 'section' is emitted.

Example
Suppose the model defines an element ''Class'' which has two other embedded elements:

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

When the part ''Class'' is generated, using the following template:

```
<_TDEBlock_>
class $name$ {            <-- this $name$ is from the Class object
  <hasscope NAME="Method">
  /*Starting Method Declaration*/
  <scope NAME="Method">
     void $name$();
 </scope>
  /*End of Method Declaration*/
```

```
 </hasscope>
};
</_TDEBlock_>
```

The following is emitted:

```
class Set {
      /*Starting Method Declaration*/
      void add();
      void del();
      /*End of Method Declaration*/
}
```

ifhasscope directive

The ifhasscope directive behaves in the same manner as a hasscope directive followed by a scope directive. This directive first tests whether the scope name exists. If the scope name does exist the scope name is navigated. This directive may be used in conjunction with an else statement.

Syntax

```
<ifhasscope NAME="sname">
sectionA
<else/>
sectionB
</ifhasscope>
```

Description

The sname specifies the name of the scope

Example

Suppose the model defines an element ''Class'' which has two other embedded elements:

```
<Class name="Set">
    <Method name="add">
    </Method>
    <Method name="del">
    </Method>
</Class>
```

When the part ''Class'' is generated, using the following template:

```
<_TDEBlock>
  class $name$ {
  <ifhasscope NAME="Method">
  /*Starting Method Declaration*/
     void $name$();
  /*End of Method Declaration*/
  <else/>
     method was not there
</ifhasscope>
};
</_TDEBlock_>
```

The following is emitted:

```
class Set {
```

```
        /*Starting Method Declaration*/
        void add();
        /*End of Method Declaration*/
    }
```

Note only the first method is emitted. In order to emit the second method a second "ifhasscope" directive must be added to the template file. If it were not possible to scope to method, the else statement would have been executed.

ifhasrepeatscope directive

The ifhasrepeatscope directive behaves in the same manner as a hasscope directive followed by a repeatscope directive.This directive first tests whether the scope name exist. If the scope name does exist all children elements that matches the scope name are navigated. This directive works in conjunction with the else statement.

Syntax

```
<ifhasrepeatscope NAME="sname">
sectionA
<else/>
sectionB
</ifhasrepeatscope>
```

Description

The sname specifies the name of the scope

Example

Suppose the model defines an element ''Class'' which has two other embedded elements:

```
    <Class name="Set">
        <Method name="add">
        </Method>
        <Method name="del">
        </Method>
    </Class>
```

When the part ''Class'' is generated, using the following template:

```
<_TDEBlock>
  class $name$ {
  <ifhasrepeatscope NAME="Method">
  /*Starting Method Declaration*/
     void $name$();
  /*End of Method Declaration*/
  <else/>
     no methods were found
</ifhasrepeatscope>
};
</_TDEBlock_>
```

The following is emitted:

```
class Set {
      /*Starting Method Declaration*/
      void add();
```

```
        void del();
        /*End of Method Declaration*/
    }
```

Note that in this example the second method is emitted. The ifhasrepeatscope navigates to all elements that matches the scope name. In this case the scope name is "Method".

Modifying Existing DOM Trees

Template files can be created to modify existing DOM trees associated with a domain model. There are several main operations that are defined to accomplish this task. Add an element, remove an element, scope to an element, update an attribute, update an element, add an attribute and remove an attribute.

targetscope directive

The targetscope directive navigates an element that is part of an existing DOM tree associated with a domain model.

Syntax

```
<targetscope NAME="sname" INDEX=_index_>
section
</targetscope>
```

where,

sname = target scope name

index = integer value

Description

The sname specifies the name of the scope. The _index_ value specifies which element to navigate based on an index value. The default value is set to 0. Therefore, if there is more than one element that is found, the first element is scoped.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Classlist>
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
   <targetscope NAME="Class">
   </targetscope>
</_TDEBlock_>
```

The mechanism will navigate to the Class element in the class.xml document. Note no emitted output is created since this directive just navigates the targeted XML document. In this case the targeted XML document is class.xml.

addtargetscope directive

The addtargetscope directive is used to insert an element into a DOM tree associated with a domain model.

Syntax

```
<addtargetscope NAME="sname" INDEX=_index_>
```

where,

sname = target scope name
index = integer value

Description

The sname specifies the name of the scope. The _index_ value specifies where to insert the element. The default value of the index is set to 0. Therefore, the element that is added will become the first child element. If a -1 is specified as the index the element that is added will be the last child element.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. Lets say that this document is called class.xml.

```
<Classlist>
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
```

```
<targetscope NAME="Class">
   <addtargetscope NAME="Method">
      <addtargetscope NAME="Argument"></addtargetscope>
   </addtargetscope>
</targetscope>
</_TDEBlock_>
```

The result of applying the template is as follows:

```
<ClassList>
<Class name="Set">
<Method>
    <Argument/>
</Method>
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
</Classlist>
```

updatetargetscope directive

The updatetargetscope directive is used to update an element of a DOM tree associated with a domain model. If the element does not exist it is inserted into the DOM tree.

Syntax

```
<updatetargetscope NAME="sname" INDEX=_index_>
```

where,

sname = target scope name

index = integer value

Description
The sname specifies the name of the scope. The default value of the index is set to 0. Therefore, the first element will be updated. The index value is only used when adding an element.

Example: Simple target document update
Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<define MACRO=newMethodName>addNew</define>
   <targetscope NAME="Method">
   <updatetargetscope NAME="name"
TYPE="ATTRIBUTE>$newMethodName$</addtargetscope>
   </targetscope>

</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
<Method name="addNew">
</Method>
<Method name="del">
</Method>
</Class>
```

Notice the name of the first method changes from "add" to "addNew".

removetargetscope directive

The removetargetscope directive is used to delete an element from a DOM tree associated with a domain model.

Syntax

```
<removetargetscope NAME="sname" INDEX=_index_/>
```

where,

sname = target scope name

index = integer value

Description

The sname specifies the name of the scope. The _index_ value specifies which element to remove based on an index value. The default value is set to 0. Therefore, if there is more than one element, the first element is removed. If the index is set to -1 the last child element is removed.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. Lets say that this document is called class.xml

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
   <removetargetscope NAME="Method"/>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
<Method name="del">
</Method>
</Class>
```

Note that the first method element was removed. Since no index was specified the default value for the index is 0.

updatetargetdoctype directive

The updatetargetdoctype directive is used to update the !DOCTYPE element of a DOM tree associated with a domain model. If the element does not exist is it inserted into the DOM Tree.

Syntax

```
<updatetargetdoctype SYSTEM_URL="SYSTEM_URL"
PUBLIC_NAME="DTD_NAME" PUBLIC_URL="DTD_URL"
ROOT_ELEMENT_NAME="root_element">
```

where,

DTD_name = a name that identifies the dtd file

DTD_URL = an absolute or relative url where the dtd file can be found

SYSTEM_URL = an absolute or relative url where the dtd file can be found

root_element = the root element tag name

Description

The DTD_URL is an absolute or relative path where the dtd file can be found. The DTD_URL is used if one wants to specify a public dtd. This attribute should only be specified if a public dtd is being used. Similarly the SYSTEM_URL is also an absolute or relative path where the dtd file can be found. This should be specified if a system dtd is being used. The DTD_name is a identifier that identifies the dtd file and is used when specifing a public dtd file. The root_element is the name of the root element of the document. If the root_element is not specified the mechanism will search the target document and find the root element tag name.

Example1: Adding a simple !DOCTYPE element

Consider the following template file:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../../dtd/tde.dtd">
<_TDEBlock_>
   <updatetargetdoctype PUBLIC_NAME="Chemistry"
PUBLIC_URL="http://sunsite/unc.edu/public/chemistry.dtd"
ROOT_ELEMENT_NAME="myroot"/>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
 <?xml version="1.0" standalone="no"?>
<!DOCTYPE myroot PUBLIC "Chemistry"
"http://sunsite/unc.edu/public/chemistry.dtd">
```

Example2: Adding a simple !DOCTYPE element plus an element.

Consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../../dtd/tde.dtd">
<_TDEBlock_>
   <updatetargetdoctype SYSTEM_URL="mydtd.dtd"
 ROOT_ELEMENT_NAME="MyMember"/>
   <updatetargetscope NAME="MyMember">
      This is the text for the new member
   </updatetargetscope>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE MyMember SYSTEM "mydtd.dtd">
<MyMember/>
```

Example3: Adding a simple !DOCTYPE element after adding an element.

Consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "../../../../dtd/tde.dtd">
<_TDEBlock_>
   <updatetargetscope NAME="MyMember">
      This is the text for the new member
   </updatetargetscope>
   <updatetargetdoctype SYSTEM_URL="mydtd.dtd"/>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE MyMember SYSTEM "mydtd.dtd">
<MyMember>
      This is the text for the new member
</MyMember>
```

Note that since the root node already exists (ie MyMember is created) at the point where "updatetargetdoctype" is defined, the ROOT_ELEMENT attribute does not need to be specified. The mechanism will know that "MyMember" is the

root element and fill in the appropriate information in the !DOCTYPE element.

addattribute directive

The addattribute directive adds an attribute to an existing element.

Syntax

<addattribute NAME="attrname">attrbute value</addattribute>

where,

attrname = attribute name.

Description

The attrname specifies the name of the attribute. If the attribute already exits the attribute will not be added to the element.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
   <addtargetscope NAME="Method">
   <addattribute NAME="name">newMethod</addattribute>
   </addtargetscope>
</targetscope>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
<Method name="newMethod">
</Method>
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

updateattribute directive

The updateattribute directive behaves in the same manner as the addattribute directive.

Syntax

```
<addattribute NAME="attrname">attrbute value</addattribute>
```

where,

attrname = attribute name.

Description
The attrname specifies the name of the attribute. If the attribute already exists the attribute value will be updated with the value specified in this directive.

Example
Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
   <addtargetscope NAME="Method">
   <updateattribute NAME="name">newMethod</addattribute>
   </addtargetscope>
   <updateattribute NAME="name">NewSetClass</addattribute>
</targetscope>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="NewSetClass">
<Method name="newMethod">
</Method>
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Note the first updateattribute directive adds an attribute named "newMethod" on the newly created method element. The second updateattribute directive updates the "name" attribute on the Class element.

removeattribute directive

The removeattribute directive removes an attribute from an existing element.

Syntax

```
<removeattribute NAME="attrname"/>
```

where,

attrname = target attribute name

Description

The attrname specifies the name of the attribute.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Method">
   <removeattribute NAME="name"/>
</targetscope>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
<Method>
</Method>
<Method name="del">
</Method>
</Class>
```

Note the "name" attribute on the first Method element was removed.

addtext directive

The addtext directive adds text to an existing element.

Syntax

```
<addtext INDEX="_index_">text content</addtext>
where,
   _index_ = integer value
```

Description

The _index_ value specifies where to insert the text. The default value of the index is set to 0. Therefore, the text content that is added will become the first child element.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
```

```
   <addtext>This class is a collection of objects</addtext>
</targetscope>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
This class is a collection of objects
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

updatetext directive

The updatetext directive behaves in the same manner as the addtext directive.

Syntax

```
<updatetext INDEX="_index_">text content</updatetext>
```

where,

index = integer value

Description

The _index_ value specifies which text content to update. The default value of the index is set to 0. Therefore, the text content that is updated will become the first child element. If no text content is found at the specified index it is added at the index specified.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
This is a collection of objects
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
   <targetscope NAME="Method">
   <updatetext>//This method adds an object to the
collection</updatetext>
   </targetscope>
   <updatetext>//This is a collection of objects. Also known as
a set of objects.</updatetext>
</targetscope>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="NewSetClass">
```

```
//This is a collection of objects.  Also known as a set of
objects.
<Method name="add">
//This method adds an object to the collection
</Method>
<Method name="del">
</Method>
</Class>
```

Note the first updatetext directive adds text content to the first method element. The second updatetext directive updates text content associated with the Class element.

removetext directive

The removetext directive removes text content from an existing element.

Syntax

```
<removetext INDEX="_index_"/>
```

where,

index = integer value

Description

The _index_ value specifies which text content to remove. The default value of the index is set to 0. Therefore, the text content that is removed will be the first child element.

Example

Suppose the model defines an element "Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
This is a collection of objects
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
   <removetext/>
</_TDEBlock_>
```

the result of applying the template is as follows:

```
<Class name="Set">
<Method>
</Method>
<Method name="del">
</Method>
</Class>
```

Note the text content under the Class element was removed.

hastargetscope directive

The hastargetscope directive behave in a similar manner as the hasscope directive. However, this directive applies to the targeted DOM tree. If the targeted DOM tree contains the scope name the section of code defined within this directive is parsed.

Syntax

```
<hastargetscope NAME="sname" INDEX=_index_>
section
</hastargetscope>
where,
   sname = target scope name
 _index_ = integer value
```

Description

The sname specifies the name of the scope. The _index_ value specifies if the element at the index value exists. The default value is set to 0.

Example

Suppose the model defines an element ''Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
   <hastargetscope NAME="NOElement"/>
   <removetargetscope NAME="Method"/>
   </hastargetscope>
</targetscope>
</_TDEBlock_>
```

this resulting DOM tree is as follows:

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Note that no changes were made. Since the "NOElement" was not found in the targeted DOM tree everything within the hastargetscope does not get processed.

repeattargetscope directive

The repeattargetscope directive behaves in a similar manner as the repeatscope directive. However, this directive applies to the targeted DOM tree. The repeattargetscope iterates over a list of child elements.

Syntax

<repeattargetscope NAME="sname">

section

</repeattargetscope>

where,

sname = target scope name

Description

The sname specifies the name of the scope.

Example

Suppose the model defines an element "Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<repeattargetscope NAME="Method">
      <addattribute NAME="type">void</addattribute>
</repeattargetscope>
</_TDEBlock_>
```

this resulting DOM tree is as follows:

```
<Class name="Set">
<Method name="add" type="void">
</Method>
<Method name="del" type="void">
</Method>
</Class>
```

ifhasrepeattargetscope directive

The ifhasrepeattargetscope directive behaves in a similar manner as the ifhasrepeatscope directive. However, this directive applies to the targeted DOM tree. The ifhasrepeattargetscope directive iterates over a list of child elements. If a child list does not exist the section of text between the directive does not get processed. This directive may also be used with the else statement.

Syntax

```
<ifhasrepeattargetscope NAME="sname">
sectionA
<else/>
sectionB
</ifhasrepeattargetscope>
```

where,

sname = target scope name

Description

The sname specifies the name of the scope.

Example

Suppose the model defines a list of elements. This document is called element.xml.

```
<ElementList>
<Element index="1"/>
<Element index="2"/>
<Element index="3"/>
<Element index="4"/>
<Element index="5"/>
<Element index="6"/>
<Element index="7"/>
<Element index="8"/>
</ElementList>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="element.xml">
   <ifhasrepeattargetscope NAME="NOElement">
      <addattribute NAME="name">ElementName</addattribute>
   <else/>
      <addtargetscope NAME="NOElement"/>
   </ifhasrepeattargetscope>
</_TDEBlock_>
```

this resulting DOM tree is as follows:

```
<ElementList>
<NOElement>
```

```
<Element index="1"/>
<Element index="2"/>
<Element index="3"/>
<Element index="4"/>
<Element index="5"/>
<Element index="6"/>
<Element index="7"/>
<Element index="8"/>
</ElementList>
```

Note the DOM tree would not change. The 'addattribute' does not get processed since the 'ifhasrepeattargetscope' is trying to scope to an element type called "NOElement" that does not exist. The else statement is then processed and a new element is added.

ifhastargetscope directive

The ifhastargetscope directive behaves in a similar manner as the ifhasscope directive. However, this directive applies to the targeted DOM tree. If the targeted DOM tree contains the scope name the section of code defined within this directive is scoped. The else directive maybe used here as well.

Syntax

```
<ifhastargetscope NAME="sname" INDEX=_index_>
sectionA
<else/>
sectionB
</ifhastargetscope>
```

where,

sname = target scope name

index = integer value

Description

The sname specifies the name of the scope. The _index_ value specifies if the element at the index value exists. The default value is set to 0.

Example

Suppose the model defines an element "Class'' which has two other embedded elements. This document is called class.xml.

```
<Class name="Set">
<Method name="add">
</Method>
<Method name="del">
</Method>
</Class>
```

Also consider the following template:

```
<?xml version="1.0"?>
<!DOCTYPE _TDEBlock_ SYSTEM "dtd/tde.dtd">
<_TDEBlock_ DOMTree="class.xml">
<targetscope NAME="Class">
   <ifhastargetscope NAME="Method"/>
      <addattribute NAME="type">void</addattribute>
   <else/>
      <addtargetscope NAME="Method"/>
   </ifhastargetscope>
```

```
      <ifhastargetscope NAME="NoMethod"/>
         <addtargetscope NAME="Element"></addtargetscope>
      <else/>
         <addtargetscope NAME="newMethod"/>
      </ifhastargetscope>
   </targetscope>
   </_TDEBlock_>
```

this resulting DOM tree is as follows:

```
   <Class name="Set">
   <Method name="add" type="void">
   </Method>
   <Method name="del">
   </Method>
   <newmethod>
   </newmethod>
   </Class>
```

Note that the attribute was added to the first method. No element changes were made. Since the "NOMethod" was not found in the targeted DOM tree, the else statement gets processed and a new element is added called newmethod.

createalias directive

The createalias directive is used to define logical roots. These logical roots acts as pointers on a DOM tree.

Syntax

```
<createalias ALIASNAME="alias" ALIASPATH="sname"/>
```

where,

sname = target scope name

alias = the alias name that represents the logical root

Description

The sname specifies the name of the scope. The alias represents the logical root. Using alias names simplifies navigating a DOM tree. Furthermore, using alias names can make the template more readable. Alias names are globally visible just like macros.

Example

Consider the following source XML document:

```
<Package name=MyPackage>
      <ClassList>
          <Class name=HondaCar>
                <Method name=addHondaCar>
                </Method>
                <Method name=removeHondaCar>
                </Method>
          </Class>
          <Class name=FordCar>
                <Method name=addFordCar>
                </Method>
                <Method name=removeFordCar>
                </Method>
          </Class>
       </ClassList>
</Package>
```

Now consider the following template file:

```
<_TDEBlock_>
   <createalias ALIASNAME="HondaClass"
ALIASPATH="ClassList//Class[0]"/>
   <scope NAME="ClassList//Class[1]">
   <createalias ALIASNAME="FordAddCarMethod"
ALIASPATH="Method[0]"/>
   </scope>
   <scope NAME="HondaClass">
          <repeatscope NAME="Method">
                $name$
         </repeatscope>
   </scope>
   <scope NAME="ClassList//Class[1]">
        $FordAddCarMethod//name$
    </scope>
</_TDEBlock_>
```

The resulting generated output is as follows:

```
addHondaCar
removeHondaCar
addFordCar
```

The above example defines two alias names. The alias name "HondaClass" defines a logical root that references the first class element in the DOM tree. The second alias, "FordAddCarMethod", references the first method under the second class. As one can see alias names can be used to

jump throughout a DOM tree. An alias name can be overridden.

Text Content

Description

An XML document consists of child elements that may or may not have a text content. The text content of a child element is represented by a macro called $TEXT$. Consider the following XML document.

```
<class>
   <Name>Foo</Name>
</class>
```

A template file can access the text contents of the 'Name' element as follows (see Scoping Directive Section):

```
<_TDEBlock_>
<scope NAME="class">
   <scope NAME="Name">
        $TEXT$
   </scope>
</scope>
</_TDEBlock_>
```

The resulting emitted code would be:

```
        Foo
```

The template file can get complicated if numerous text contents need to be accessed. A scoping rule for each sibling would need to be implemented. In order to improve usability a scoping operator can be used. This scoping operator is defined as '//'. Therefore the above template file can be rewritten as follows:

```
<_TDEBlock>
<scope NAME="class">
       $Name//Text$
</scope>
<_TDEBlock_>
```

Positional navigation that is used in scoping can also be used with macros. This is useful in cases where there exists elements of the same type. This is illustrated in the following example:

```
<class>
    <Name>Foo</Name>
    <Name>Foo2</Name>
</class>
```

The following template illustrates positional navigation.

```
<_TDEBlock_>
<scope NAME="Class">
     $Name[2]//TEXT$
     $Name[1]//TEXT$
</scope>
</_TDEBlock_>
```

The emitted code is as follows:

```
Foo2
Foo
```

If no index is given the first sibling that matches the macro is used. Similarly, each element can has a list of text contents. Consider the following XML document.

```
<class>
    Extra Text1
    <Name>Foo</Name>
    Extra Text2
    <Method>MyMethod</Method>
    Extra Text3
</class>
```

Each text content under an element is given an index value. This index identifies each text content so that positional navigation can be used to navigate the text content list. This is illustrated in the following template file.

```
<_TDEBlock_>
<scope NAME="class">
    Name=$Name//Text$
    First Text =$Text[1]$
    Second Text =$Text[2]$
    Third Text =$Text[3]$
</scope>
</_TDEBlock_>
```

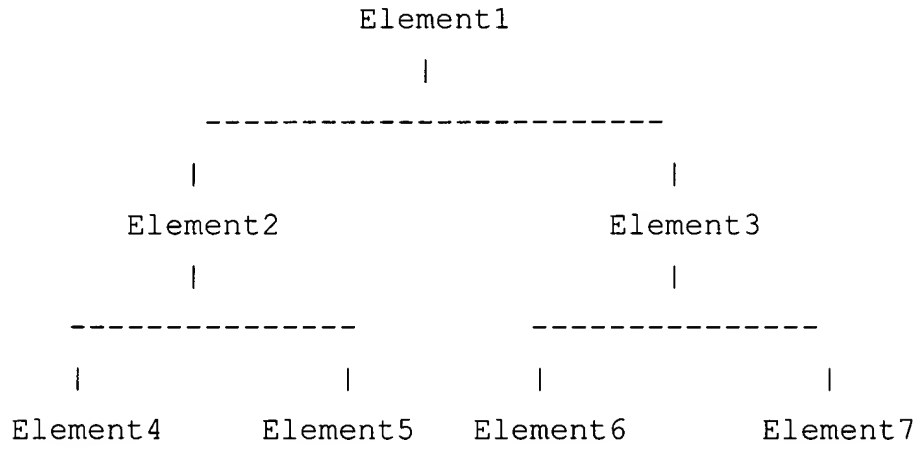
The resulting output is emitted:

```
    Name=Foo
    First Text =Extra Text1
    Second Text =Extra Text2
    Third Text =Extra Text3
```

If no index is given all text content within the element is appended together.

Children Navigation

The mechanism provides means to navigate an elements children nodes generically. For example consider the following tree:



One might want to navigate to all the end nodes of the tree without the hassle of knowing Element2 and Element3. Or one might want to repeat on all the children nodes of an element. Typically, in order to process Element2 and Element3 a separate scoping directive would have to be written for each element. For example consider the following template file:

```
<_TDEBlock_>
    <scope NAME="Element2">
          $Name$
    </scope>
    <scope NAME="Element3">
          $Name$
    </scope>
</_TDEBlock_>
```

Note for both Elements we are extracting the Name attribute. The template file can be huge if there were more children elements. Just like there is a predefined string to represent a parent for an element (ie. "..") there should be a predefined string to represent the children elements (ie. _TDECHILDREN_). With this in mind the above template can be rewritten as follows:

```
<_TDEBlock_>
    <repeatscope NAME="_TDECHILDREN_">
        $Name$
     </repeatscope>
<_TDEBlock_>
```

One can also navigate to a specific child node based on index. For example consider the following template file:

```
<_TDEBlock_>
    <scope NAME="_TDECHILDREN_[1]">
        $Name$
     </scope>
<_TDEBlock_>
```

Tag Name Content

Description

The mechanism provides a predefined macro to get the tag name of an element. This macro is called $_TDETAGNAME_$.

Consider the following XML document.

```
<class>
   <Name>Foo</Name>
</class>
```

A template file can access the tag name content of the 'Name' element as follows:

```
<_TDEBlock_>
<scope NAME="class">
   <scope NAME="Name">
        $_TDETAGNAME_$
   </scope>
</scope>
</_TDEBlock_>
```

The resulting emitted code would be:

```
        Name
```

Consider a list of students that each have a list of marks as represented by the following XML document:

```
<student_list>
    <student>
          <id>166443</id>
          <name>Sean Watt</name>
          <marks>
```

```
                <biology>63</biology>
                <math>75</math>
                <chemistry>62</chemistry>
                <physics>66</physics>
                <english>78</english>
            </marks>
        </student>
        <student>
            <id>4457833</id>
            <name>Whitney Lee</name>
            <marks>
                <biology>88</biology>
                <math>78</math>
                <chemistry>82</chemistry>
                <physics>95</physics>
                <english>81</english>
            </marks>
        </student>
        <student>
            <id>7641133</id>
            <name>Bob Barney</name>
            <marks>
                <biology>73</biology>
                <math>92</math>
                <chemistry>83</chemistry>
                <physics>76</physics>
                <english>83</english>
            </marks>
        </student>
</student_list>
```

Suppose all the biology marks and math marks are desired. We can write the following template to extract this information from the above document:

```
<_TDEBlock_>
    <scope NAME="student_list">
            <repeatscope NAME="student">
                    <scope NAME="marks">
                    <repeatscope NAME="_TDECHILDREN_">
                            <if
EXPRESSION="(($_TDETAGNAME_$==&quot;biology&quot;) |
($_TDETAGNAME_$==&quot;math&quot;))">
                                    $TEXT$
                            </if>
                    </repeatscope>
                    </scope>
            </repeatscope>
     </scope>
</_TDEBlock_>
```

The above template file navigates down to the mark element and then navigates through all the children elements under the mark element. An if statement matches the tag name of each child element. If any of the child elements are equal to "math" or "biology" the text content of the child element is extracted.

The resulting emitted code is as follows:

63
75
88
78
73
92

Comments

Comments help makes the template files easier to read. They do not get emitted in the output file. A blank line on the other hand will get emitted in the output file.

Syntax

```
<!--  comment -->
```

Example

The following are some comment lines in the template file.

```
<!--
 This is a template file
-->
```